

Requested Patent: GB2350534A

Title:

PACKET-BASED NETWORK DEVICE WITH FORWARDING DATABASE HAVING A
TRIE SEARCH FACILITY ;

Abstracted Patent: GB2350534 ;

Publication Date: 2000-11-29 ;

Inventor(s):

JENNINGS KEVIN (IE); LOUGHRAN KEVIN (IE); O'MALLEY EDELE (IE);
O'CALLAGHAN SORCHA (IE) ;

Applicant(s): 3COM CORP (US) ;

Application Number: GB19990025517 19991029 ;

Priority Number(s): GB19990012129 19990526 ;

IPC Classification: H04L12/56 ;

Equivalents:

ABSTRACT:

(12) UK Patent Application (19) GB (11) 2 350 534 (13) A

(43) Date of A Publication 29.11.2000

(21) Application No 9925517.6

(22) Date of Filing 29.10.1999

(30) Priority Data

(31) 9912129

(32) 26.05.1999

(33) GB

(71) Applicant(s)

3Com Corporation
(Incorporated in USA - Delaware)
5400 Bayfront Plaza, Santa Clara,
California 95051-8145, United States of America

(72) Inventor(s)

Kevin Jennings
Edele O'Malley
Sorcha O'Callaghan
Kevin Loughran

(74) Agent and/or Address for Service

Bowles Horton
Felden House, Dower Mews, High Street,
BERKHAMSTED, Herts, HP4 2BL, United Kingdom

(51) INT CL⁷

H04L 12/56

(52) UK CL (Edition R)

H4P PPS

(56) Documents Cited

EP 0551243 A2 WO 99/13619 A2 WO 95/34155 A2

(58) Field of Search

UK CL (Edition R) H4K KTK , H4P PPEC PPS

INT CL⁷ H04L 12/46 12/56

Online Databases: WPI, EPODOC, JAPIO

(54) Abstract Title

Packet-based network device with forwarding database having a trie search facility

(57) A communication device such as a router or switch for a packet-based network has a hardware trie search facility. At least part of a memory is divided into blocks of different sizes, each block consisting of a multiplicity of locations. A shift register holds a network address representing the destination of a packet. This network address constitutes the search key for the memory. A fixed number of bits of the key are used to access a location in a first block which provides a pointer to a second block and an indication of the size of that block. The size of the block determines the number of bits from the key required to access the block. The shift register shifts the key to the left by the number of bits previously used so that successive numbers of bits in the key are used to access successive blocks until a pointer points to an entry in an associated data table identifying a route to the required destination.

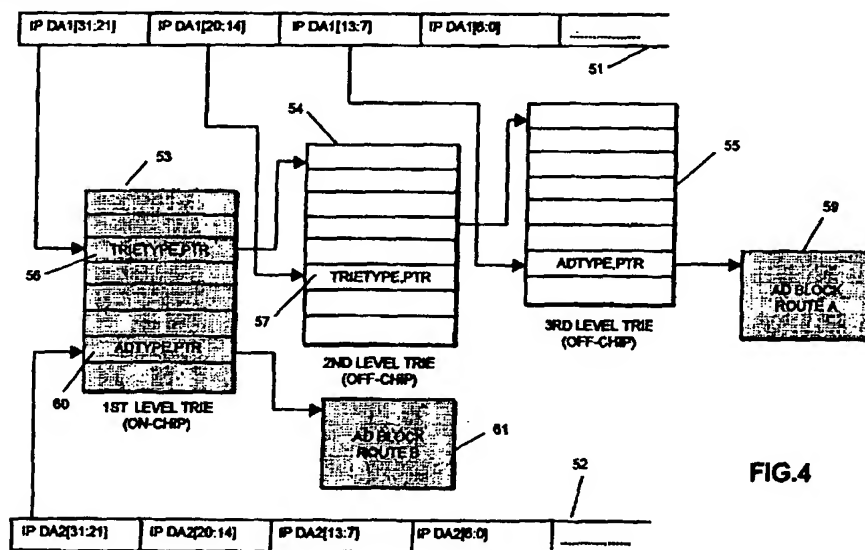


FIG.4

The reference to figure 5 of the drawings in the printed specification is to be treated as omitted under section 15(2) or (3) of the Patents Act 1977

GB 2 350 534 A

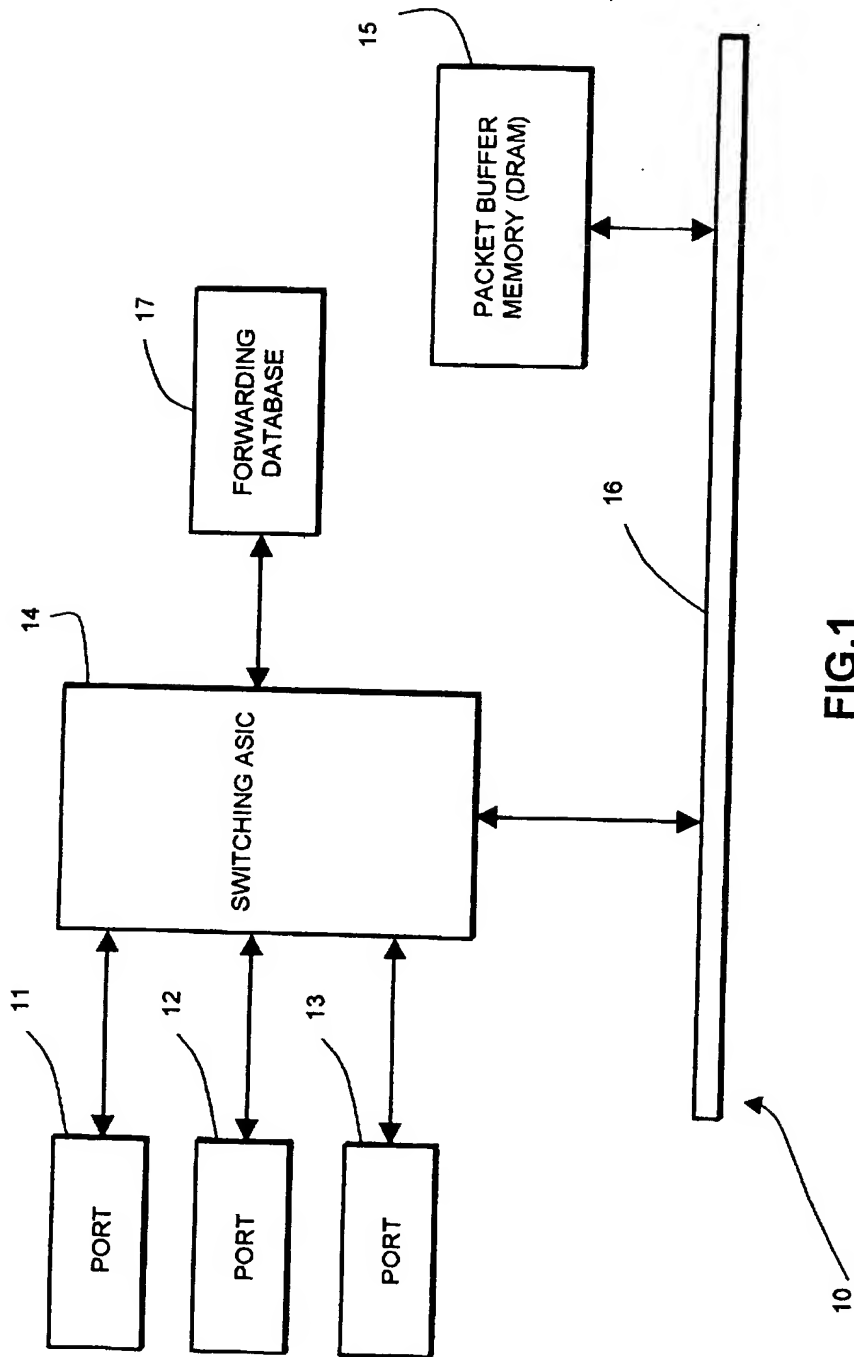


FIG.1

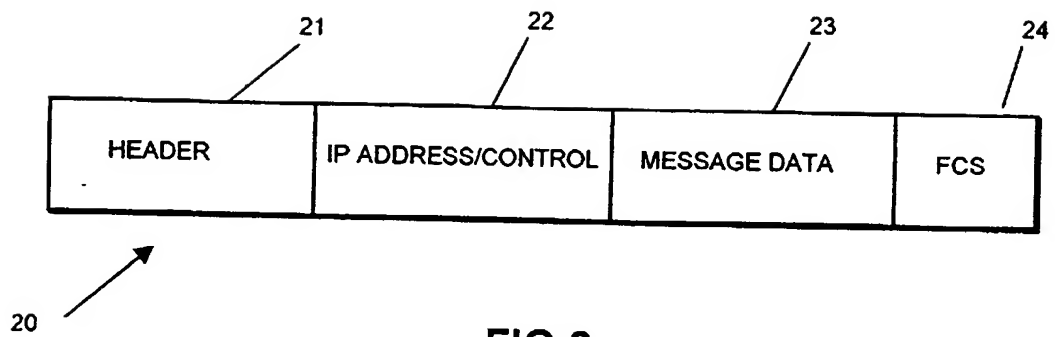


FIG.2

[illegible]

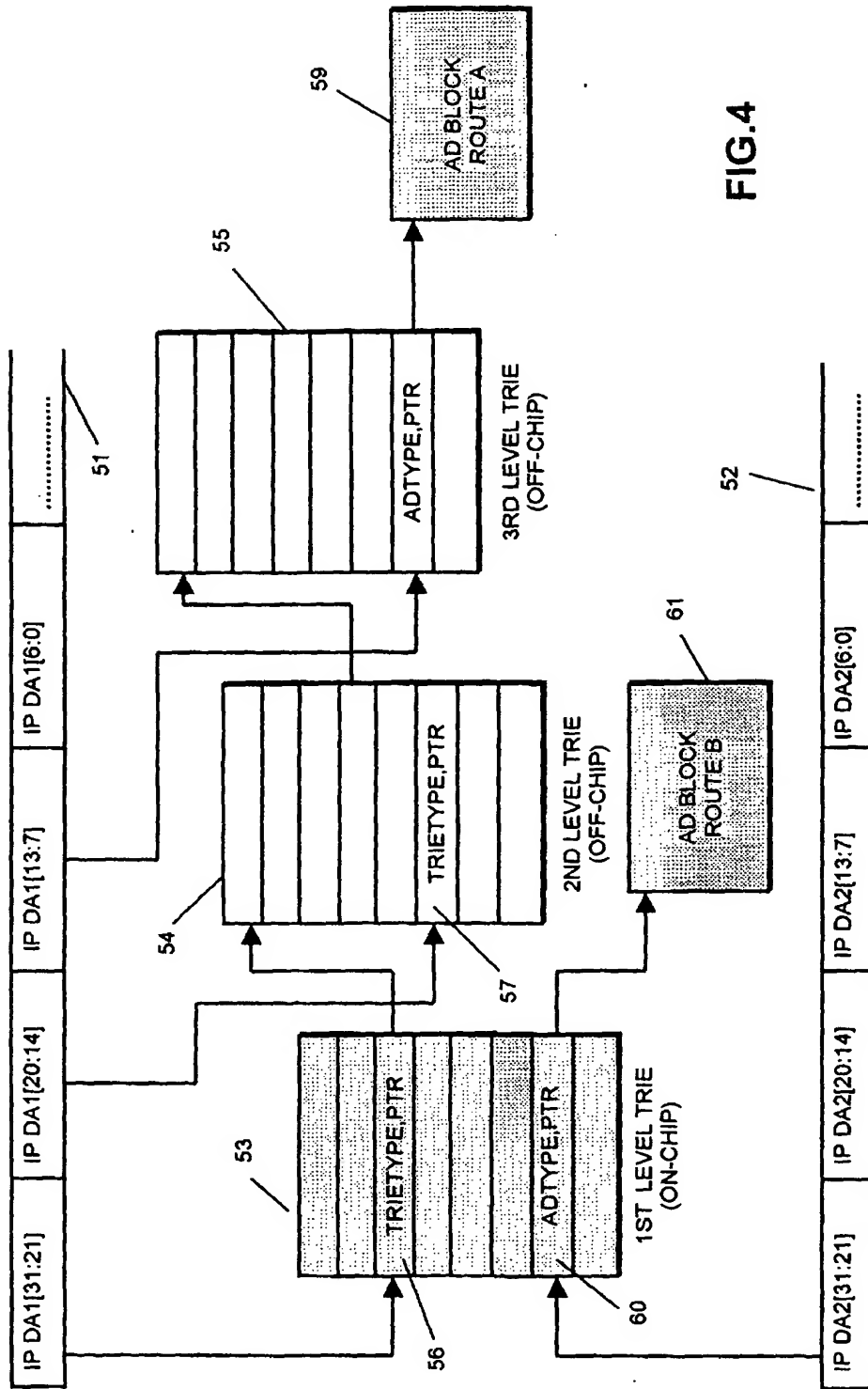


FIG. 4

COMMUNICATION DEVICE WITH FORWARDING DATABASE HAVING A TRIE
SEARCH FACILITY

5 This invention relates to communication devices for packet-based communication systems, and in particular to multi-port communication devices which receive addressed data packets at any of a multiplicity of ports and which can forward packets to one or more ports after recourse to a forwarding database which includes entries identifiable from a destination address and including associated data such as, for example, a port mask.

10 Multi-port communication devices for packet-based communication systems, such as 'layer 3' switches and routers, usually include a forwarding database by means of which a received packet is directed, after appropriate processing and temporary storage as necessary or appropriate, to one or more ports of the device. The forwarding database may be established from information obtained from routing and address resolution protocols.

15 When a packet is received and is therefore to be forwarded by the device, the destination address may be looked up in the forwarding database. There is a variety of ways of organising such a database in the look up process, in order to economize on the fast random access memory that is required and to reduce the searching time or both. For example, one
20 method of retrieval includes the hashing of an address to provide access to a table of hashed addresses, this table having entries constituting pointers to a data table including address data and associated data. It is known to organise these databases such that the entries in the data table which share a common hashed address are linked, it being necessary to verify the entries in the data table in turn against the key, or address for which the search is made, and to
25 proceed along the chain of linked addresses until a verified match is found. A database organised on these lines is disclosed in GB patent application No. 9906963.5 (publication No. 2337659).

30 There are circumstances in which despite such space saving schemes as hashing, a forwarding database is liable to contain a very large number of addresses and consequently would require a very large memory. One example is if a switch is used as a 'layer 3' device, namely it is

required to look up addresses which pertain to layer 3 (logical link control) of the OSI model, these addresses are otherwise known as 'network' addresses.

5 It is known to employ a 'trie' system for the forwarding database. The purpose of a 'trie' scheme is to determine the 'next hop' for a packet on a route to its final destination. It will be understood that a router which operates on network addresses is indirectly connected (by way of other routers) to a large number of end stations and, in the absence of some more efficient scheme for storing addresses, would need to store the MAC addresses of all that large number of end stations (in addition to the MAC addresses of the end stations to which it may be
10 directly connected). However, many packets having different network addresses will require to be sent to the same router and therefore, as far as the search is concerned may be treated as having the same address (whether this be expressed as a port number or as the MAC address of the next router). The address data for forwarding packets to the next (intermediate) destination can be collapsed into a single entry in a routing table.

15 It is therefore convenient to divide a forwarding database, particularly for a router or other layer 3 switch, into generally two parts. One part comprises a set of blocks each of which contains a plurality of storage locations each including (in the simplest case) a pointer 'result' field. The rest of the database comprises 'associated data' entries, such as a port number
20 and/or a MAC address. If the 'result' field of a pointer is not set, the pointer will identify another trie block. If it is set it will point to an entry in the associated data table part of the database. Many packets may have the same 'next hop' and accordingly the information for the address data of these packets can be collapsed into a single entry in the routing table. The address look up is performed using a key comprising the layer 3 address information. The key
25 may be used to search a trie data structure until a leaf node, namely an associated data block is found. Each level in the trie employs a successive group of bits from the key. The benefit of such a searching scheme is that many addresses with a common prefix can refer to the same target address

30 When a trie structure is established, the amount of memory allotted to it in the database is divided into blocks. The number of blocks determines the number of routes which may be

stored. If only a small number of blocks is available, then the number of routes that can be stored is also small. It is possible to increase the number of routes stored by sub-dividing the same memory space into smaller trie blocks. Using smaller blocks in the trie system has the disadvantage that the search time may be increased, thereby reducing the performance of the look up process. Maximum versatility can be achieved by supporting blocks of different sizes. Initially, the tables can then be built using larger trie blocks, maximizing look up performance. If the memory configuration does not allow sufficient routes to be stored, the tables can be configured to use smaller trie blocks with some loss of look up performance.

However, if blocks of different sizes are to be used, a problem arises over the selection of the correct number of address bits for use in each stage of the trie search. The multiplexing required to select different numbers of address bits at each stage would become very complex, the complexity increasing as the number of block sizes increases.

The invention has as its main object the provision of a device with a trie search facility that is provided in hardware and which accommodates a multiplicity of block sizes for the trie search.

In one aspect the invention provides a communication device for a packet-based communication system wherein each packet includes address data, the communication device including a multi-stage trie search system comprising: a memory; means for accessing at least part of the memory in blocks, each of said blocks consisting of a multiplicity of locations, at least some of said locations each providing when accessed a pointer to another block and an indication of the size of that block; a shift register for holding an address key; and means for accessing said another block using an address composed of a number of bits related to said indication of the size of the block and for shifting said key by said number of bits prior to a subsequent stage of the search.

In another aspect of the invention provides a method of performing a search in a database in a device for use in a packet-based data communication system, said database being organised in a multiplicity of blocks, a first plurality of blocks containing entries composed of pointers

to other blocks and a second plurality of said blocks containing entries defining a route for a packet, the blocks in said first plurality being of different sizes, said method comprising at each stage of searching: (i) employing a first plurality of bits of a search key to access a first block and identify a respective pointer therein, said respective pointer containing a result field, an address field and a size field; and (ii) when said result field has a particular value, employing a second plurality of bits all subsequent to said first plurality of bits to access a second block in the first plurality of blocks, said size field indicating the number of bits in said second plurality of bits.

The invention is based on the use of a shift register to hold the key for the trie search, and at each stage of the search to shift the content of the register by the number of bits employed for accessing a location within a block. The result of each stage of a search is a pointer to a block, the pointer identifying the block and also including an indication, which denotes the size of the block to which the pointer refers. This indication in the pointer can be used to control the shift register. Preferably the indication is a coded indication, i.e. it merely distinguishes between the possible sizes of blocks. Thus it may be constituted by a one-bit field if there are only two possible sizes and a two-bit field if there are four possible sizes. Herein 'size' refers to the number of different pointers that can be accessed and not necessarily the number of locations, since each location may be used for storing a plurality of pointers.

Reference will hereinafter be made to the accompanying drawings, which illustrate schematically one embodiment of the invention.

Figure 1 is a schematic illustration of a known form of network device.

Figure 2 is a simplified illustration of a data packet.

Figure 3 is a schematic drawing illustrating the performance of a trie search according to the invention.

Figure 4 is a further illustration of a trie search according to the invention.

The present invention is intended in its preferred form to be implemented in a router, or network device able to perform a routing function, that is to say to obtain from a packet its 'IP' or layer 3 destination address and by means of a forwarding database to select a route to the device identified by the layer 3 address. This route may be expressed as the next hop, i.e. it requires a MAC address or port number to be obtained, using the network address as a key, from the forwarding database. The architecture of the device in which the invention is to be incorporated is not important and there are many proprietary network devices which are quite adequate for hosting a trie search facility according to the invention.

Figure 1 illustrates in simplified schematic form only a network device which can be used as a host for the present invention. In common with most other network devices, the device includes a multiplicity of ports. For the sake of simplicity, only three ports, 11, 12 and 13 are shown. Such ports are commonly organised so that they have a receive path and a transmit path so that they can both receive packets from a device to which the respective port is connected and send packets from that port. The packet reading, encapsulation and switching functions are commonly performed in an ASIC (application-specification-integrated circuit) 14 which can direct receive packets to a packet buffer 15, preferably but not necessarily implemented in dynamic random access memory. Packets reach the packet buffer by way of a bus 16. Also shown in Figure 1 is a forwarding database 17 which, as is well known and as previously indicated, is used to determine a route that a packet should take in order to get to the required destination.

The forwarding database (or 'routing table') is established according to known techniques using appropriate address resolution protocols. This is beyond the scope of the present invention and will not be described further.

Figure 2 illustrates in simplified terms an Ethernet data packet of the kind employed in a communication network system in which a network device according to the invention would form part. It is not intended to be a detailed illustration of a packet, which is fully described in any of the publications relevant to the known transmission standards. Typically, a packet comprises a header 21, which may include the MAC addresses of the immediate source and

destination of the packet, a segment which includes IP address data and other control data of no consequence to the present invention, message data 23 and frame check sum or cyclic redundancy code data which is generated employing the data content of the packet and some generating functions.

5

Figure 3 illustrates schematically the main hardware elements of a search facility according to the invention. It comprises a memory 30, which is organised as described hereinafter, a shift register control 31, and a shift register shown in three phases, 32a – 32c, illustrating an initial phase 32a when a search key is loaded and two later phases. In each cycle of operation, a selected number of bits, starting with the most significant bit, of the key access the memory 30. The result is a pointer which, among other things, defines the number of bits used in the next stage of the search. The control 31 responds to this pointer by shifting the keyword held in the shift register by the number of bits employed in the current stage of search and selects the number of bits to be employed in the next stage, as will be described in more detail in what follows.

10

15

The search key, constituted by the 'network address' representing the destination of a packet which is to be forwarded from the device, is written into the shift register in phase 32a. It is presumed that the search key can have a maximum of N bits, the search register having at least this number of bit positions and the search key being written into the shift register with the most significant bit (MSB) first. The first probe into the forwarding database is into a block which is fixed in size and is therefore accessed by a fixed number of address bits. It will be assumed that this number of bits is K bits. The address is generated by using the appropriate number (K) of bits from the key. When the result of the first read of the memory is returned, the shift register control 31 shifts the key to the left by the same, fixed, number (K) of bits.

20

25

The result of the first read operation is a pointer 40 which includes a (one bit) 'result' field 41. If this field is set (i.e. is '1') the address data 42 in the pointer points to an entry in the associated data table. If the result field is not set the pointer points to another trie block. In the latter case the pointer includes address data 42 to identify the next trie block and an indication, which may be constituted by a single or multi-bit field 43, of the size of the block to which the

30

pointer points. In the present example it will be assumed that the memory employs blocks of two different sizes, 32 bytes or 256 bytes. If the block size is 32 bytes (containing eight locations each 32 bits wide, with two pointers in each location) then the next three bits of the key are used to index into the correct location within that block and a fourth bit to select the upper or lower bits in that location, to obtain the correct pointer. If the block size is 256 bytes, with 32 locations each 32 bits wide and two pointers per location, the next seven bits of the key are required to select a unique pointer. This scheme requires only a single multiplexer 45 to select either the next four or seven most significant bits of the remaining portion of the key from the same fixed location (the top seven bits) within the shift register.

When the pointer accessed by the combination of the block pointer and the relevant number of bits from the key is read the result is a pointer for yet another block and the shift register for the key must again be shifted to the left by either four or seven bits, depending on how many bits were used.

Figure 4 shows pointer 40 pointing to a block 46 which is accessed (to obtain the next pointer) by shifted key 32b. This block is shown as part of memory space 30a which could be part of memory 30 but which may be provided 'off-chip' whereas memory 30 may be 'on-chip'.

The search will continue using the next pointer 47 until the required route or next hop has been found. Depending on how the address is stored in memory, this may mean that the entire key has not to be used before the answer is found.

In order to support more than two different block sizes, the number of bits in the trie pointer must be increased. For example, to support four different block sizes, containing for example 16, 32, 64 and 128 pointers respectively, the block size field in the block pointer requires at least two bits to indicate the size of the next block. This is shown in the Figure, wherein the size field consists of a two bit field 43 to enable the shift register control to provide the shift of the corresponding number of bits and to control a multiplexer 45 to select the respective number of bits from the uppermost seven bits of the shift register.

If for example the first stage pointers are stored in 'on-chip' memory 30 to speed up the search and all further stage pointers are stored in 'off-chip' memory, a further multiplexer 48 may be provided to enable a choice between a pointer coming from memory on the chip or a pointer coming from memory 30a which is 'off-chip'.

5

The memory may be organised differently, with only one pointer per memory location but the same principles apply. In such a variation the size of the memory in terms of pointers would be equal to the size in number of locations.

10

Figure 5 illustrates two trie searches which are performed in response to two network address keys shown at 51 and 52 respectively. The searches are conducted through blocks 53, 54 and 55 which are of a different size. Block 53 requires eleven bits for access, namely bits 31 to 21 of a keyword. The second block 54 requires seven bits, as does the third block 55.

15

Figure 5 shows the search process performed in response to the key 51. Access to the first block requires the first eleven bits [31:21] of the IP destination address (key) 51 and identifies a trie type pointer 56, which points to block 54 and determines a block size of seven bits. The second stage of the search identifies a specific pointer 57 within block 54. This is again a trie type pointer which points to block 55. The third stage of a search finds a destination pointer 58 which points to an associated data block 59 identifying a route A to the destination.

20

In contrast, the search made using the first eleven bits of key 52 identifies in block 53 a pointer 60 of which the result field is set. Thus this pointer will identify an entry in an 'associated data' block 61, this entry defining the next hop on which the packet containing the IP address key 52 should be sent in the first block 53, pointing to an associated route A shown at 54.

25

30

Claims

5 1. A communication device for a packet-based communication system wherein each packet includes address data, the communication device including a multi-stage trie search system comprising:

a memory;

10 means for accessing at least part of the memory in blocks, each of said blocks consisting of a multiplicity of locations, at least some of said locations each providing when accessed a pointer to another block and an indication of the size of that block;

a shift register for holding an address key; and

15 means for accessing said another block using an address composed of a number of bits related to said indication of the size of the block and for shifting said key by said number of bits prior to a subsequent stage of the search.

20 2. A device according to claim 1 wherein said indication of the size of the block is a coded indication of the number of accessible pointers and accordingly the number of bits required to identify a pointer within a respective block.

25 3. A device according to claim 1 or 2, wherein a first stage of the trie search is in a block of a predetermined size and the key in the shift register is shifted by a number of bits corresponding to that predetermined size in preparation for a second stage of the trie search.

30 4. A device according to any foregoing claim wherein part of the memory comprises an associated data table which includes entries of data enabling a packet to be forwarded and wherein at least some locations in said blocks provide when accessed a pointer to an entry in the associated data table.

5. A method of performing a search in a database in a device for use in a packet-based data communication system, said database being organised in a multiplicity of blocks, a first plurality of blocks containing entries composed of pointers to other blocks and a second plurality of said blocks containing entries defining a route for a packet, the blocks in said first plurality being of different sizes, said method comprising at each stage of searching:

(i) employing a first plurality of bits of a search key to access a first block and identify a respective pointer therein, said respective pointer containing a result field, an address field and a size field; and

(ii) when said result field has a particular value, employing a second plurality of bits all subsequent to said first plurality of bits to access a second block in the first plurality of blocks, said size field indicating the number of bits in said second plurality of bits.

6. A method according to claim 5 and further comprising, when said first result field has a predetermined value different from said particular value, employing said respective pointer to access a block in said second plurality of blocks.



Application No: GB 9925517.6
Claims searched: All

II

Examiner: Gareth Griffiths
Date of search: 25 April 2000

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK CI (Ed.R): H4K (KTK), H4P (PPEC, PPS)

Int CI (Ed.7): H04L 12/46, 12/56

Other: Online Databases: WPI, EPODOC, JAPIO

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
A	EP0551243 A2 (DIGITAL EQUIPMENT)	
A	WO99/13619 A2 (SICS SWEDISH INSTITUTE)	
A	WO95/34155 A2 (NOKIA)	

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

Requested Patent: EP0419889A2
Title: PREFIX SEARCH TREE WITH PARTIAL KEY BRANCHING. ;
Abstracted Patent: EP0419889 ;
Publication Date: 1991-04-03 ;
Inventor(s): NICKEL STEVEN P (US) ;
Applicant(s): BULL HN INFORMATION SYST (US) ;
Application Number: EP19900116841 19900903 ;
Priority Number(s): US19890414045 19890928 ;
IPC Classification: G06F15/413 ; G06F15/419 ;
Equivalents:

AU6099990, AU641420, CA2022970, CN1050630, JP2670383B2, JP3122766,
KR9612665, US5202986

ABSTRACT:

A prefix index tree structure for locating data records stored through keys related to information stored in data records. Each node includes a prefix field for a prefix string of length p of the longest string of key characters shared by all subtrees of the node and a data record field for a reference to a data record whose key is completed by the prefix string. A node may include one or more branch fields when the prefix string is a prefix of keys stored in at least one subtree of the node, with a branch field for each distinct $p+1$ key character in the keys, wherein each $p+1$ key character is a branch character. Each branch field includes a branch character and a branch pointer field for a reference to a node containing at least one key whose $p+1$ character is the branch character. Each node further includes a field for storing the number of key characters in the prefix string and a field for storing the number of branch fields in the node. Also disclosed are methods for constructing and searching a prefix index tree of the present invention, and for inserting nodes into the tree and deleting nodes from the tree.



Europäisches Patentamt
European Patent Office
Office européen des brevets



0 419 889 A2

EUROPEAN PATENT APPLICATION

⑤¹ Int. Cl.⁵: **G06F 15/413**, **G06F 15/419**

②② Date of filing: 03.09.90

72 Inventor: Nickel, Steven P.
50 Reservoir Street
Cherry Valley, Mass. 01611(US)

74 Representative: **Altenburg, Udo, Dipl.-Phys. et al**
Patent- und Rechtsanwälte
Bardehle-Pagenberg-Dost-Altenburg
Frohwitter-Gelssler & Partner Postfach 86 06 20
W-8000 München 86(DE)

⑧ Designated Contracting States:
DE ES FR GB IT

**(71) Applicant: Bull HN Information Systems Inc.
Corporation Trust Center 1209 Orange Street
Wilmington Delaware(US)**

⑤4 Prefix search tree with partial key branching.

(57) A prefix index tree structure for locating data records stored through keys related to information stored in data records. Each node includes a prefix field for a prefix string of length p of the longest string of key characters shared by all subtrees of the node and a data record field for a reference to a data record whose key is completed by the prefix string. A node may include one or more branch fields when the prefix string is a prefix of keys stored in at least one subtree of the node, with a branch field for each distinct $p + 1^{\text{st}}$ key character in the keys, wherein each $p + 1^{\text{st}}$ key character is a branch character. Each branch field includes a branch character and a branch pointer field for a reference to a node containing at least one key whose $p + 1^{\text{st}}$ character is the branch character. Each node further includes a field for storing the number of key characters in the prefix string and a field for storing the number of branch fields in the node. Also disclosed are methods for constructing and searching a prefix index tree of the present invention, and for inserting nodes into the tree and deleting nodes from the tree.

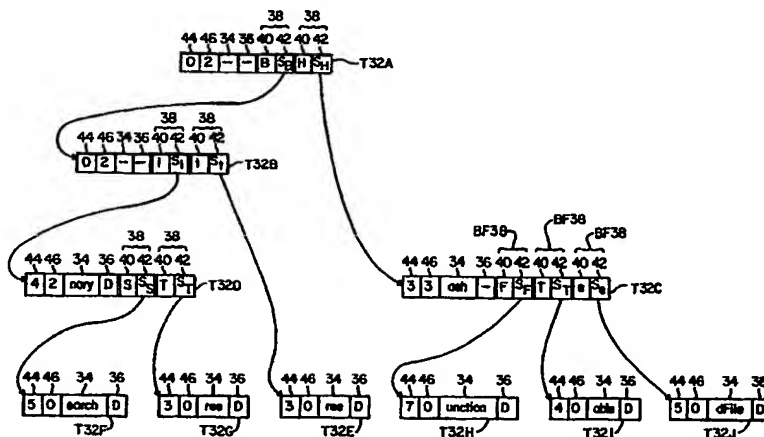


Fig 3

PREFIX SEARCH TREE WITH PARTIAL KEY BRANCHING

Cross References to Related Applications

5

Background of the InventionField of Use

10

The present invention relates generally to the indexing, or location, of information in a database through the use of keys and, in particular, to a prefix search tree for indexing a database.

15 Prior Art

A recurring problem in databases, in particular those implemented in computer systems, is the search for and location of specific items of information stored in the database. Such searches are generally accomplished by constructing a directory, or index, to the database, and using search keys to search
20 through the index to find pointers to the most likely locations of the information in the database.

In its most usual forms, the index to a database is structured as a tree comprised of one or more nodes connected by branches. Each node generally includes one or more branch fields containing information for directing a search, wherein each such branch field usually contains a pointer, or branch, to another node, and an associated branch key indicating ranges or types of information may be located along that branch
25 from that node. The tree, and any search of the tree, begins at a single node referred to as the root node and progresses downwards through the various branch nodes until the nodes containing either the items of information or, more usually, pointers to the items of information are reached. The information related nodes are often referred to as leaf nodes, or, because this is the level at which the search either succeeds or fails, failure nodes. It should be noted that any node within a tree is a root node with respect to all nodes
30 dependent from that node, and such sub-structures within a tree are often referred to as sub-trees with respect to that node.

The decisions as to what directions, or branches, to take through a tree in a search is determined, at each node encountered in the search, by comparing the search key or keys and the branch keys stored in the node. The results of the comparisons determine which of the branches depending from a given node
35 are to be followed in the next step of the search. In this regard, search keys are most generally comprised of strings of characters or numbers which relate to the item or items of information to be searched for. For example, "search", "tree", "trees" and "search tree" could be keys to search a database index for information relating generally to search trees while "617" and "895" could be keys to find all telephone numbers in the 895 exchange of the 617 area. The forms taken by the branch keys depend upon the type
40 of search tree, as described briefly below.

The prior art contains a variety of search tree structures, among which is the apparent ancestor from which all later tree structures have been developed, and the most general form of search tree, the "B-tree". A B-tree is a multi-way search tree wherein each node is of the form $(A_0K_0) \sim (A_1K_1) \sim \dots \sim (A_nK_n)$ and wherein each A_i is a pointer to a subtree of that node and each K_i is a key value associated with that subtree. All key
45 values in the subtree pointed to by A_i are less than the key value of K_{i+1} , all key values in subtree A_n are greater than K_n , and each subtree A_i may also be a multi-way search tree. The decision as to which branch to take at a given node is performed by comparing the search key K_x to the branch keys K_i of the node and following the pointer A_i associated with the lowest value key K_i which is larger than K_x ; the search will follow pointer A_0 if K_x is less than all keys K_i and will follow pointer A_n if K_x is greater than key K_n .

50 The next variant on the basic B-tree is the Binary Tree wherein each node is of the general form (A_i, K_i, A_{i+1}) . Each node of a Binary tree therefore contains only one branch key and two branches, so that there are only two ("binary") branches from any node. The leftmost branch A_i is taken if search key K_x is less than node key K_i and the rightmost branch A_{i+1} is taken if search key K_x is greater than K_i .

The B'-tree and the B*-tree are similar to the B-tree except that in the B'-tree all information or pointers to information may be located only in the leaf nodes, that is, the lowest nodes of the tree, while in the B*-

tree all failure nodes, that is, all leaf nodes, are at the same level in the tree. The B*-tree also has specific requirements on the maximum and minimum number of branches depending from the root and branch nodes.

The Bit Tree is again similar to the B-tree in its root and branch nodes, but differs in its leaf nodes in that the Bit Tree does not store keys in the leaf nodes. Instead, each pointer in a leaf node has associated with it a "distinction bit" which indicates the first bit in which the key for that branch differs from the branch key contained in the root, or next higher, node to that leaf node. Distinction bits are generated by comparing the binary expression for the branch key for a pointer in a leaf node with the binary expression for the node key of its root node and noting the binary number of the lowest order bit in which the two keys differ. That number, which is actually the number of the distinction or difference bit, is then stored in the leaf node in association with the pointer. A search is conducted, at the leaf node level, by comparing the search key with the node key of the leaf's parent node and determining the lowest order bit in which the search key differs from the node key; the search then takes the leaf's pointer which is associated with the next lower order distinction bit.

The Trie is an index tree using variable length key values and wherein the branching at any level of the Trie is determined by only a part of the key, rather than by the whole key. Also, in a Trie the branching at any level is determined by the corresponding sequential character of the key, that is, the branching at the j^{th} level of the trie is determined by the j^{th} character of the key. Searching a Trie for a key value K_n requires breaking K_n into its component characters and following the branching values determined by those component characters. If, for example, the $K_n = \text{LINK}$, then the branching at the first level is determined by the branch corresponding to component L, at the second level by component I, at the third level by N, and at the fourth level by K. This requires that, at the first level, all possible characters of the search keys be partitioned into individual, disjoint classes, that there be a first level branch for each class, and that the Trie contain a number of levels corresponding to the number of characters in the longest expected search key.

Finally, in a Prefix B-tree each node is again of the form $(A_0K_0) \sim (A_1K_1) \sim (A_nK_n)$ and is searched in the same manner as a B-tree, but each key K_i in a Prefix B-tree is not a full key but is a "separator", or prefix to a full key. The keys K_i of each node in any subtree of a Prefix B-tree all have a common prefix, which is stored in the root node of the subtree, and each key K_i of a node is the common prefix of all nodes in the subtree depending from the corresponding branch of the node. Again, there is a binary variant of the Prefix B-Tree, referred to as a Prefix Binary Tree, in which each node contains only one branch key and two branches, so that there are only two ("binary") branches from any node. The Prefix Binary Tree is searched in the same manner as a Binary Tree, that is, branching left or right depending on whether the search key is less than or greater than the node key. There are also, in turn, Bit Tree variants of the Prefix Binary Tree wherein distinction bits rather than prefixes are stored in the nodes. In particular, the values stored are the numbers of the bits in the keys which are different between two prefixes, thus indicating the key bits to be tested to determine whether to take the right or left branches.

The above described search trees of the prior art are generally intended to provide certain optimum characteristics for the most general cases of information searches and the most general types or classes of information. Certain trees may be designed, for example, to provide the minimum depth of tree so as to reduce the number of disk accesses required to bring successive nodes or groups of nodes into system memory, or to provide the minimum search time, or to equalize the search times for all searches, or to allow the easy insertion or deletion of nodes. The tree structures of the prior art do not, however, provide optimum structures for certain broad classes of information. For example, the prior art tree structures are generally not optimum in cases wherein the keys may be divided into rather large partitions, as is the case with certain types of information, and do not provide the optimum structures for creating and modifying search trees for such types of keys and information.

Yet another disadvantage of the tree structures of the prior art is that it is generally necessary to search completely to the data record level to determine whether or not a particular data item is present in the database. This is often described as a requirement that all failure nodes be at the same level in the tree. This disadvantage arises from the inherent search methodology as determined by the structure of the trees. As described, the search key is compared to the node keys to determine the branch paths having the range of key values most likely to contain a match with the search key. Because the search is based upon identifying the branches having ranges of key values, there is no point in the search short of the actual data records that a determination can be made as to whether a search key can actually be matched to a data record.

A solution to the above described problems of the prior art, and other problems, are provided by a prefix index tree of the present invention which is particularly adapted to those classes of information wherein the keys may be divided into rather large partitions. The tree structure of the present invention

further provides an improved structure for creating and modifying search trees for such types of keys and information. The tree structure of the present invention further does not require that all searches continue to the data record level before it can be determined that a particular data item is not present in the database.

5

Summary of the Invention

The tree structure of the present invention provides a prefix index tree structure for locating data records stored in a database in a data processing system through keys related to the information stored in the data records. Each node of the tree includes a prefix field for storing a prefix string of length p comprised of the longest string of key characters shared by all subtrees of the node and a data record field for storing a reference to a data record whose key is completed by the prefix string. The tree structure further includes one or more branch fields when the prefix string is a prefix of keys stored in at least one subtree of the node. There is a branch field for each distinct $p+1^{\text{st}}$ key character in the keys of the subtrees, wherein each distinct $p+1^{\text{st}}$ key character is a branch character. Each branch field includes a branch character field for storing the $p+1^{\text{st}}$ character of a key and a branch pointer field for storing a reference to a node of a subtree containing at least one key whose $p+1^{\text{st}}$ character is the branch character.

In further embodiments of the present invention, each node further includes a field for storing a number equal to the number of key characters in the prefix string, and a field for storing a number equal to the number of branch fields in the node.

The present invention further includes methods for constructing and searching a prefix index tree of the present invention, and for inserting nodes into the tree and deleting nodes from the tree.

25

Brief Description of the Drawings

The foregoing and other objects, features and advantages of the present invention will be apparent from the following description of the invention and embodiments thereof, as illustrated in the accompanying figures, wherein:

Fig. 1 is a diagrammatic representation of a data processing system and an index tree resident therein;

Fig. 2 is a diagrammatic representation of a node of a tree of the present invention;

Fig. 3 is a diagrammatic illustration of a tree of the present invention;

Figs. 4A, 4B and 4C are illustrations of the insertion of nodes into a tree; and,

Fig. 5 is an illustration of the deletion of nodes from a tree.

40

Description Of The Preferred Embodiments

A. General Description of a Tree in a Data Processing System (Fig. 1)

Referring to Fig. 1, therein is an illustrative representation of a Data Processing System 10 and an Index Tree 12, with Tree 12 arranged to illustrate the residence of Tree 12 in the addressable memory space of System 10. System 10 is comprised of a Central Processing Unit (CPU) 14, which is in turn comprised of an Arithmetic and Logic Unit (ALU) 16 with associated Working Registers 18, a directly addressable Memory 20, which may also include a cache memory, and associated storage in the form of a Disk 22.

Tree 12 is represented as having a single Root Node 24 and a plurality of Branch Nodes (Node) 26 and Leaf Nodes (Leaf) 28, all connected through Pointers, or branches, 30. As indicated, the Branch Nodes 26 are further designated according to their levels in Tree 12, that is, according to their depth in Tree 12 and, correspondingly, the number of nodes that must be traversed to reach a given node. In this illustration of a tree, there are two Level 1 Branch Nodes, each designated as a L1 Node 26, several Level 2 and Level 3 Branch Nodes, each respectively designated as a L2 Node 26 or a L3 Node 26, and a single Level 4 Branch Node, designated as L4 Node 26.

Tree 12 is positioned relative to System 10 in Fig. 1 to illustrate the locations of the various elements of Tree 12 in System 10's address space, and arrows extend rightwards from System 10 to indicate the

boundaries of the various regions of System 10's address space. For example, at the start of a search, as illustrated in Fig. 1, Root Node 24 would most probably be located in Working Registers 18 of System 10's CPU 14, and thus would be directly accessible to ALU 16, as would one or more of the L1 Nodes 26 and possibly one or more of L2 Nodes 26. Further of Nodes 26 and perhaps certain of Leafs 28 would be found in Memory 20, while the deeper nodes of Tree 12 would be found stored as files in Disk 22.

The locations of the various Tree 12 nodes in System 10's address space effects the specific forms taken by the nodes and by the Pointers 30 stored therein. For example, and as will be described in the following detailed description of a Tree 12 according to the present invention, each node is always of the same basic form, that is, a set of fields containing specific types of information in a specific format. Nodes residing in Working Registers 18, however, are located in specific registers while nodes located in Memory 20 reside in physical memory locations which may be dynamically reassigned and which are located through logical addresses. Nodes residing in Disk 22 will reside in disk files. Correspondingly, the Pointers 30 to nodes residing in Working Registers 18 may take the form of logical address pointers, or more likely, specific ALU 16 register identifications. Pointers 30 to nodes located in Memory 20 will take the form of logical address pointers which are translated, by System 10, to Memory 20 physical addresses when their corresponding nodes are to be accessed. Pointers 30 to nodes residing in Disk 22 will be in the form of file references. It should be noted, however, that while the specific forms of the information contained in the fields of a node may change with the location of the node in System 10's address space, the functional and structural and logical relationships of the various elements of the nodes of Tree 12 remain the same.

The locations of the nodes in System 10's address space also affect the speed with which System 10 may access the nodes and process the information contained therein, and correspondingly the speed with which System may perform a search. For example, the nodes residing in Working Registers 18 are directly accessible to ALU 16 and may be processed in correspondingly little time. The nodes residing in Memory 20 and in any associated cache memory are also relatively quickly accessible to CPU 14, requiring only the delay of a logical to physical address translation and a memory access cycle to be read into Working Registers 18 as the search progresses. The access time to the nodes of Tree 12 become greater, however, the deeper into Tree 12 the search progresses. In particular, the nodes residing in Disk 22 require a disk access operation and a file read to be transferred into Memory 20, and a subsequent transfer into Working Registers 18. It is therefore advantageous that Tree 12 be as "flat" as possible, that is, contain as high a degree of branching as possible, to move the nodes up towards the root node to decrease the node access time, and, in particular, reduce the number of disk accesses required to search Tree 12. It is also advantageous to move the Leaf Nodes 28 up into Tree 12's structure as far as possible, rather than requiring all Leaf Nodes 12 to reside at the same, and lowest, level of Tree 12. As will be described next below, the Tree 12 of the present invention provides an approach to providing these advantages for certain broad classes of information.

B. Description of a Tree of the Present Invention (Figs. 2 and 3)

A Tree 12 of the present invention is designed for use wherein the keys may be placed into suitably large partitions determined by leading characters shared with other keys. Tree 12 is a dense index structure using variable length, character oriented keys. Branching at any level is determined by a part of the key, rather than by the whole key, and the structure of the Tree 12 is independent of the order in which the Tree 12 is constructed.

A Tree 12 of the present invention is a prefix search tree that is either empty or is of height greater than or equal to one, that is, contains one or more levels, and satisfies the following properties:

- (i) Any node, T, of the tree is of the form and type $p, s, (P_1, \dots, P_p), D, ((B_1, S_1), \dots, (B_s, S_s))$ where the P_i , $0 < i \leq s$, represent the prefix string, the tuples (B_i, S_i) , $0 < i \leq s$, are branch characters and subtrees of T, respectively, and D is a pointer to a data record;
- (ii) The prefix (P_1, \dots, P_p) contains the longest string of leading characters shared by every key contained in T (and the subtrees dependent from T);
- (iii) D is a pointer to the record with the key of length p, or is a null if there is no such key;
- (iv) Each B_i , $0 < i \leq s$, is a distinct character which is the $p + 1^{\text{st}}$ character of some key in T, that is, of a subtree dependent from T, whose length is greater than p;
- (v) $B_i < B_{i+1}$, $0 < i < s$;
- (vi) Each S_i is a pointer to a prefix search tree dependent from T; and,
- (vii) The keys in a subtree referenced by a S_i , $0 < i \leq s$, are formed from the set of keys in T having B_i as

their $p+1^{\text{st}}$ character, by removing their initial $p+1$ characters.

Referring to Fig. 2, therein is presented a diagrammatic illustration of the structure and format of a single node (T) 32 of a Tree 12 of the present invention according to the definition presented above. As shown, T 32 may contain a Prefix Field (PF) 34 which contains a prefix of length p ($P_1...P_p$) comprised of the longest string of characters shared by all keys of every subtree dependent from node T 32, and a Data Pointer Field (D) 36 which contains a Pointer 30 to a data record having the key ($P_1...P_p$), if there is such a key and data record. T 32 may also contain one or more Branch Fields (BFs) 38, each of which is comprised of a Branch Character Field (BC) 40 for storing a branch character B_j and a Branch Pointer Field (BP) 42 for storing a corresponding branch pointer S_j . As described, each B_j is the $p+1^{\text{st}}$ character of a key of length greater than p of a subtree dependent from T 32 while each associated S_j is a pointer to the node T 32 of that subtree. Finally, each node T 32 will include a p Field 44 and an s Field 46 containing, respectively, the length, or number of characters, in the prefix stored in PF 34 and the number of subtrees (or data records) dependent from the node T 32, that is, the number of BF 38's contained in the node T 32. Although p Fields 44 and s Fields 46 are not a necessary part of the structure of nodes T 32, these fields are provided to assist System 10 in processing the nodes. That is, it is more efficient to inform the processor as to the length of the prefixes contained in the PF 36s and the number of Branch Fields 38 than to have the system extract this information from the PF 36s and BF 38s.

As will be described below with reference to Fig. 3, certain nodes of a Tree 12 of the present invention may be "leaf" nodes, which are identical in structure to the branch nodes T 32 except that they contain no Branch Fields 38 as the branches are nulls.

Referring to Fig. 3, therein is a diagrammatic illustration of a Tree 12 of the present invention using the key values "Btree", "Binary", "BinarySearch", "BinaryTree", "HashTable", "HashFunction", and "HashedFile".

It is apparent from an examination of the keys used for this example that the Tree 12 of Fig. 3 will have two branches, or subtrees, dependent from the root node. One branch will contain nodes for the keys having the initial character "B" (Btree, Binary, BinarySearch, and BinaryTree) and other for the nodes for the keys having the initial character "H" (HashTable, HashFunction and HashedFile). Accordingly, PF 34 of root node T 32A will be null as there is no common prefix shared between the keys starting with "B" and the keys starting with "H", and T 32A's D field 36 will also be a null as there are no data records dependent from T 32A. T 32A will contain a first BF 38 field for the T 32A subtree containing all keys having an initial character "B" and a second BF 38 field for those keys having initial character "H". Considering the first BF 38 field, the BF 40 field B_j character in this field will be the character "B" as "B" is the $p+1^{\text{st}}$ character of the keys of the corresponding subtree of T 32A and the BP 42 field will contain an S_j pointer S_B to the first node in this subtree, T 32B. The second BF 38 field of T 32A will contain the character "H" as its B_j in the BC 40 field as this is the $p+1^{\text{st}}$ character of the keys of the corresponding subtree, and the S_j pointer in the BP 42 field will be a pointer S_H to the first node in this subtree, T 32C. The p field 44 and s field 46 of T 32A will respectively contain a 0 to indicate that the PF 34 field of T 32A contains no prefix characters, that is, is a null, and a 2 to indicate that T 32A has two "children", that is, that there are two branches from T 32A.

Considering T 32B, the next branch in the keys having initial character "B" will occur between the key "Btree", having "t" as its second character, and the keys having "i" as their second character (Binary, BinarySearch and BinaryTree). There are no common prefix characters shared between the keys branching from this node, so that T 32B's PF 34 field will contain a null, as will T 32B's D field 36. The T 32B will again have two BF 38s, with the first having a B_j of "i" and the second having a B_j of "t", "i" and "t" being the $p+1^{\text{st}}$ characters of the keys of the subtrees dependent from these branches. The corresponding S_j pointers will be pointers S_i and S_t to, respectively, nodes T 32D and T 32E. The p Field 44 and s Field 46 of T 32B will respectively contain a 0, indicating that the PF 34 field contains no prefix characters, and a 2, indicating that T 32B has two children, or branches.

Next considering T 32E, this node contains a reference to a data record, but no further branches to further nodes. As such, the PF 38 fields of T 32E contain nulls, that is, the node contains no PF 38 fields. The PF 34 field of T 32E contains the final portion of the key for the associated data record, the character string "ree" in the case of T 32E, and a D field 36 containing a pointer to the data record. The p Field 44 and s Field 45 respectively contain a 3, indicating that the PF 34 field contains three characters, and a 0, indicating that Leaf 48A has no branches to subtrees.

Next considering T 32D, the other node dependent from node T 32B, the subtree of which T 32D is the root node contains the keys "Binary", "BinarySearch" and "BinaryTree", wherein the prefixes "B" and "i" of these keys are stored as prefixes in the PF 34 fields of, respectively, T 32A and T 32B. The longest prefix common to the remaining portions of these keys, that is, to "nary", "narySearch" and "naryTree" is the character string "nary". As such, the character string "nary" is stored as a prefix in the PF 34 field of T

32D.

Of the three keys in this subtree, all three keys differ in the next character following "nary" and T 32D could thus have three branches. "nary" is, however, the final portion of the key "Binary", so that, rather than resulting in a branch to another node, the key "Binary" results in a pointer to the data record associated with the key "Binary" being written into the D field 36 of T 32D.

The keys "BinarySearch" and "BinaryTree", however, have remaining character strings following "nary" and thus result in branches from T 32D. The $p+1^{\text{st}}$ character of "BinarySearch" is "S", so that "S" appears as the B_j of a first BF 38, together with an S_j pointer S_s to the associated node T 32F in the BP Field 42. The $p+1^{\text{st}}$ character of "BinaryTree" is "T", so that "T" appears as the B_j of the second BF 38, together with an S_j pointer S_T to the associated node T 32G in the BP Field 42. The p Field 44 and s Field 46 of T 32D respectively contain a 4, to indicate that the PF 34 field contains a string of 4 characters, and a 2, to indicate that there are two branches from T 32D.

T 32F and T 32G are both similar to T 32E in that these nodes contain no further branches to other nodes, and thus have null, or empty, BF 38 fields, but pointers to associated data records in their respective D 36 fields. The PF 34 field of T 32F contains the character string "earch", which is the final portion of the key "BinarySearch", while the PF 34 field of T 32G contains the character string "ree", which is the final portion of the key "BinaryTree". The p Field 44 of T 32F contains a 5, for the five characters in "earch" and the p Field 44 of T 32G contains a 3, for the three characters in "ree", while the s Field 46 of each node contains a zero, indicating that there are no branches from either node.

Referring briefly to the right hand subtree of Tree 12, comprised of nodes T 32C, T 32H, T 32I and T 32J, this subtree is constructed by the same principle as just described above. The keys contained in this subtree are "HashTable", "HashFunction" and "HashedFile" and the character "H" of all three keys appears as the B_j of the corresponding PF 38 of T 32A as the $p+1^{\text{st}}$ character of the prefix appearing in PF 34 of T 32A. As previously described, PF 34 of T 32A contains a null character string as there is no common prefix character string between the two branches dependent from T 32A.

The longest prefix string common to the remaining portions of these keys, that is, to "ashTable", "ashFunction" and "ashedFile" is the string "ash" and "ash" accordingly appears in the PF 34 field of T 32C. Because there are three keys having a the common prefix string "ash", there will be three branches from T 32C. The $p+1^{\text{st}}$ characters of the remaining portions of these three keys are, after removing "ash", respectively, "T", "F" and "e". "T", "F" and "e" accordingly appear as the B_j s in the BF 38s of T 32C, together with corresponding S_j pointers S_T , S_F and S_e to nodes T 32H, T 32G and T 32I. The p Field 44 and s Field 46 of T 32C respectively contain a 3, to indicate a character string of three characters in PF 34, and a 3, to indicate that there are three branches from T 32C.

Nodes T 32H, T 32G and T 32I are again "leaf" nodes in that they contain pointers to data records in their D fields 36, but no further branches and correspondingly no BF 38s. The PF 34 field of T 32G contains the string "unction", which is the remaining portion of key "HashFunction", while the PF 34 fields of T 32G and T 32H respectively contain "able" and "dFile", the final portions of keys "HashTable" and "HashedFile". The s Fields 46 of each of these nodes contain 0s, as there are no branches from these nodes. The p Fields 44 of these nodes respectively contain a 7, a 4 and a 5, representing the number of characters in the remaining portions of the keys stored in their PF 34 fields.

C. Searching of a Tree 12

In order to search for any given key value in the Tree 12 of the present invention, System 10 begins at the root node and proceeds through the Tree 12, node by node, as described in the following, until the search reaches a failure node, that is, a node which has no match for the search key, or succeeds by finding the data record corresponding to the search key.

Starting in the root node, the system compares the search key (K), which has a length, or number of characters, k, to the prefix character string (P), which has a length p, stored in the PF 34 of the node to determine whether the prefix matches at least the initial characters of the search key. That is, to determine whether $K \geq P$ and $K_i = P_i$ for some $i \leq p$. In this regard, it should be noted that if the prefix $P = 0$, that is, if P is a null string, then zero characters of the search key and prefix are considered matched.

If there is a complete match between search key K and prefix P, that is, $P = K$, then the corresponding data record is pointed to by the pointer stored in the D field 36 of the node.

If there is a match between the prefix character string, which has a length p, and the first p characters of the search key character string, then the system searches the B_j s of the BC 40 fields of the BF 38's to find a B_j which matches the $p+1^{\text{st}}$ character of the key K (K_{p+1}). If the search finds no $B_j = K_{p+1}$, then the

key value is not contained in the node and the search has failed.

If the search finds a $B_j = K_{p+1}$, then the search follows the associated S_j pointer to the corresponding next node and continues the search. It will be remembered, however, that the prefix for each succeeding node in the tree is comprised of the longest prefix string common to the remaining portions of the keys after removal of the leading prefix characters which have been incorporated into the prefixes of previous nodes. In a like manner, the key used to search a next node of the tree has a new key value of $K_{p+2}..K_k$, that is, is comprised of the portion of the search key remaining after removal of the leading key characters which have been matched to prefixes and branch characters in previous nodes.

Further description of the searching of a tree of the present invention may be found in the following exemplary Search Program Listing A:

15

20

25

30

35

40

45

50

55

PROGRAM LISTING A - TREE SEARCH

```

5  procedure PSEARCH (T, (K1..Kk))
      // Search the prefix search tree T residing on
      disk for the key value (K1..Kk). A tuple
      (i,d) is returned; i is false if K does not
10     exist. Otherwise i is true and d is the data
      record pointer //
      if (T=0) then return(FALSE,0) // special case: tree
15     is empty //
      X=T; n=0
      loop
20         input node X from disk
         let X define p,s,(P1..Pp),D,((B1,S1)
           ..(Bs,Ss))
25         // if the prefix is too long, can't possibly
           match the key //
         if n+p>k then return(FALSE,0)
         // match the prefix to the leading characters
30         in the key //
         for i=1 to p do
             n=n+1
35             if Kn<>Pi then return (FALSE,0)
         end
         // determine if this node contains the key //
40         if n=k then (if D=null then return(FALSE,0)
           else return (TRUE,D))
         // determine which node to process next.
           Search branch characters //
45         n=n+1
         j=1
         loop
50

```

55

```

      case
      :j>s:return(FALSE,0)
      :Kn<Bj:return(FALSE,0)
5      :Kn=Bj:exit
      :else:j=j+1
      end
10      forever
      X=Sj
      forever
15      end PSEARCH

```

20 D. Construction of a Tree and Insertion of Nodes (Figs. 4A, B and C)

The construction of a Tree 12 is performed in and by the same manner and method as is used to insert new nodes into an existing tree, except that the initial node of a new tree is inserted into an otherwise empty tree. For this reason, the following discussion will describe the insertion of nodes into an existing tree, with the understanding that the description applies equally to the construction of new trees.

25 There are five general conditions requiring the insertion of a new node into a Tree 12:

- (a) A mismatch occurs between a prefix and a new key before the end of either character string, a condition referred to as a "prefix collision";
- (b) A new key is longer than the prefix in question and the key matches for the entire length of the prefix but there are either no branch characters or the next character in the key after the last character of the prefix is not among the branch characters, a condition referred to as a "branch collision";
- 30 (c) A new key is shorter than the prefix in question, and the prefix and the key match for the entire length of the key, a condition referred to as an "initial substring";
- (d) The length of a new key is equal to that of the prefix in question, and the key and the prefix match, but there is no data associated with the prefix, a condition referred to as a "data collision"; and,
- 35 (e) The tree is empty.

Considering first the instance of a prefix collision, a prefix collision requires the creation of three nodes to replace the node where the collision occurred; one to replace the previously existing node and two nodes dependent from that node. Of the two new dependent nodes, one will contain the portion of the key occurring beyond the character which caused the match to fail and the other will contain the portion of the prefix occurring beyond the character which caused the match to fail. The third node, which is the replacement for the original node, will contain the portion of the original prefix which matched with the key and will include two branches and, correspondingly two BF 38s. One BF 38's B_j will be the character of the prefix which caused the match to fail and the associated S_j will point to the new subnode containing the remaining portion of the original prefix. The other BF 38's B_j will be the character of the key which caused the match to fail, and the associated S_j will point to the new subnode containing the remaining portion of the key.

This operation is illustrated in Fig. 4A, wherein the new key "HashTable" is to be added to a tree at a node T 48A containing the prefix "HashFunction". The initial character strings "Hash" of the original prefix and the new key match, but the match fails at the "F" of the original prefix and the "T" of the new key. A first new subnode T 48B is created whose PF 34 contains the portion of the original prefix occurring after the prefix failure character, that is, the string "unction" which follows the prefix failure character "F". Original node T 48A had a D Field 36 pointer to a data record, so that new first subnode T 48B also has a D Field 36 pointer to that same data record. If node T 48A had contained a Field BF 38, this would then appear in the new subnode T 48B.

55 The second new subnode T 48C contains in its PF 34 the portion of the key occurring after the key failure character, that is, the string "able" which follows the key failure character "T". Second new subnode T 48C will also contain a D Field 36 pointer to the data record associated with the key "HashTable".

Finally, new node T 48D which replaces original node T 48A has the string "Hash" in its PF 34, that is,

the portion of the prefix and key strings which matched. A first BF 38 of new node T 48D contains a B_j of "F", that is, the prefix character which failed in the match, and an associated S_j pointer to the new subnode having the prefix "unction", the remaining portion of the original prefix. A second BF 38 of new node T 48D contains a B_j of "T", that is, the key character which failed in the match, and an associated S_j pointer to the new subnode having the prefix "able", the remaining portion of the key. Although original node T 48A had a D Field 36 pointer to a data record, this pointer now appears in first subnode T 48B, so that this replacement for original node T 48A has no D Field 36 pointer.

Next considering the case of a branch collision, a branch collision requires the creation of two nodes to replace the original node where the collision occurred. One node will be a subnode which will contain in its PF 34 the portion of the key occurring beyond the character which was not found among the branch characters B_j of the original node. The other new node will contain the prefix, branch characters and subtrees of the original node in which the branch collision occurred, with the addition of a branch character B_j, the new branch character being the key character which was not found as a branch character in the original node. Associated with this new branch character will be an S_j pointer to the new subnode.

This operation is illustrated in Fig. 4B, wherein the new key "HashedFile" is to be added to the tree resulting from the operation illustrated in Fig. 4A. The new key "HashedFile" is longer than prefix "Hash" of node T 48D and matches the entire prefix. The next character of the key, "e", however, is not found in the BF 38s of T 48D. Accordingly, a new node T 48E is created containing, as a prefix in its PF 34, the key character string "dFile", which is the portion of the key after the unfound branch character "e". A corresponding new BF 38 is created for T 48D with branch character "e" and an associated S_j pointer to new node T 48E. It should be noted that new node T 48E contains a D Field 36 pointer to the data record associated with key "HashedFile" and that nodes T 48B and T 48C remain unchanged.

Considering the instance of an initial substring, when an initial substring is encountered two nodes are created to replace the node where the collision was detected. The first node will contain, in its PF 34, the portion of the prefix which was not matched by the key, minus its initial character, together with the subtrees and branch characters of the original node. The other node will contain, in its PF 34, the portion of the prefix which was matched by the key, with the initial character of the unmatched portion of the key as its sole branch character and an associated S_j pointer to the first node, which will be a subnode of this second node.

This operation is illustrated in Fig. 4C, wherein the key "Binary" is to be added to node T 48F which has prefix "BinarySearch" and a D Field 36 pointer to a data record. The "Binary" characters strings of both the key and the prefix match, while the "Search" portion of the prefix is not matched by the key. Accordingly, a new node T 48G is created having the string "earch" as its prefix, that is, the portion of the original prefix which was not matched by the key, minus its initial character, "S". T 48G also has a D Field 36 pointer to the data record originally associated with original node. If T 48F had had branch characters and branch pointers to other nodes of the tree, these branch characters and pointers would be replicated in the new node T 48G. The second new node T 48H is created with a prefix of "Binary", that is, the portion of the original prefix which was matched by the key, and a single branch character "S", which is the initial character of the portion of the original prefix which was not matched by the key. Associated with branch character "S" will be a S_j pointer to the new node T 48G and T 48H will contain a D Field 36 pointer to any data record associated with the key "Binary".

Finally, there are the cases of a data collision and an empty tree. As described, in a data collision the length of a new key is equal to the length of the prefix and the key and prefix match but there is no data associated with the prefix. Data collisions are handled simply by adding the data to the node and rewriting the node with a D Field 36 pointer to the data record.

The instance of an empty tree is similarly straightforward. The system creates an initial node by selecting a suitable root node prefix for the tree, for example, by selecting a set of keys providing the longest common prefix, and proceeds to add further nodes according to the methods described above.

Further description of the above node insertion methods will be found in the following exemplary Insert Program Listing B:

PROGRAM LISTING B - NODE INSERT

```

5  procedure PININSERT(T,(K1..Kk),d)
      // Insert the key value (K1..Kk) into the
      prefix search tree T, with data record pointer
      d. False is returned if d is null or if the
10  key value already exists. Otherwise, true is
      returned //
      if(d=null) then return(FALSE) // special case: d is
15  null //
      if(T=null) // special case: tree
      is empty //
20  then (T=MAKENODE((K1..Kk),d,()); return
      (TRUE))

      X=T;Y=null;y=0;n=0;j=0
25  loop
      input node X from disk
      let X be defined by (P1..Pp),D,
30  ((B1,S1)..(Bs,Ss))
      // match the prefix to the leading
      characters in the key //
35  l=MIN(p,k-n)
      for i=1 to l do
          n=n+1
          if Kn<>Pi then return (PREFIX(d,
40  n,(K1..K1),i,X,Y,Y))
      end
      // is the new key a subset of an existing
45  key? //
      if n=k then {
          if l=p then {
50

```

55

```

    if D<>null then return(FALSE)
    // trivial case; replace null
    pointer with d //
5      D=d; output X to disk; return
      (TRUE) )
      return(SUBSTRING(d,n,(K1..Kk),
10      l+1,X,Y,Y)) )
    // determine which node to process next.
    Search branch characters //
15      n=n+1
      y=j;j=1
      loop
20          case
            :j>s:return(BRANCH(d,n,(K1..
              Kk),j,X,Y,Y))
            :Kn<Bj:return(BRANCH(d,n,
25              (K1..Kk),j,X,Y,Y))
            :Kn=Bj:exit
            :else:j=j+1
30          end
      forever
      Y=X;X=S
35  forever

```

40

45

50

55

INSERT FOR PREFIX COLLISION

```

5  procedure PREFIX(d,n,(K1..Kk),i,X,Y,Y)
      // a collision has occurred within the prefix
      // portion of a node. Three new nodes will be
      // formed, U, V, and W, replacing the node in
10     which the conflict occurred, X. Kn and Pi
      // were the conflicting characters.
      // y is the subtree in Y, the parent node of X,
15     which points to X //

      // assume X,Y are already in memory
      let X define p,s,(P1..Pp),D,((B1,S1)
20         ..(Bs,Ss))
      let Y define Yp,Ys,(YP1..YPp),YD,
25         ((YB1,YS1)..(YBs,YSs)) //

      // create new node U to hold remainder of new key
      // and its data //
30     U+MAKENODE((Kn+1..Kk),(d),())
      // create new node V to hold remainder of prefix
      // and subtrees //
35     V=MAKENODE((Pi+1..Pp),(D),((B1,S1)
        ..(Bs,Ss)))
      // create new node W to hold common prefix and new
      // subtrees //
40     if Kn<Pi
        then W=MAKENODE((P1..Pi-1),(),((Kn,U),
45         (Pi,V)))
        else W=MAKENODE((P1..Pi-1),(),((Pi,V),
50         (Kn,U)))

```

55


```

//  replace pointer to X in Y with pointer to W,
//  then destroy X //
5   if Y=null
      then T=W
      else (YSY=W;output Y to disk)
10  KILLNODE(X); return(TRUE)
end PREFIX
```

15

20

25

30

35

40

45

50

55

INSERT FOR BRANCH COLLISION

```

5  procedure BRANCH(d,n,(K1..Kk),j,X,y,Y)
      // a collision has occurred within the branch
      // portion of a node. Two new nodes will be
      // formed, U and W, replacing the node in which
10  the conflict occurred, X. Kn was the
      character not found in (B1..Bs).

15  j provides the insertion point. y is the
      subtree in Y, the parent node of X, which
      points to X //

20  // assume X,Y are already in memory
      let X define p,s,(P1..Pp),D,((B1,S1)
          ..(Bs,Ss))
25  let Y define Yp,Ys,(YP1..YPp),YD,((YB1,
          YS1)..(YBs,YSs)) //

30  // create new node U to hold remainder of new key
      and its data //
      U=MAKENODE((Kn+1..Kk),(d),())
35  // create new node W to hold remainder of prefix
      and subtrees //
      W=MAKENODE((P1..Pp),(D),((B1,S1)..(Bj-1,
40  Sj-1),(Kn,U),(Bj,Sj)..(Bs,Ss)))
      // replace pointer to X in Y with pointer to W,
      then destroy X //
      if Y=null
45  then T=W
      else {YSy=W;output Y to disk}
      KILLNODE(X);return(TRUE)
50  end BRANCH

```

55

INSERT FOR INITIAL SUBSTRING

```

5      procedure SUBSTRING(d,n,(K1..Kk),i,X,Y,Y)
      // an underflow has occurred within the pre-
      // fix portion of a node. Two new nodes will
      // be formed, V and W, replacing the node in
10     // which the key was exhausted, X. Pi
      // WOULD be the next character examined. y
      // is the subtree in Y, the parent node of X,
15     // which points to X //

      // assume X,Y are already in memory
      let X define p,s,(P1..Pp),D,
20     ((B1,S1)..(Bs,Ss))
      let Y define Yp,Ys,(YP1..YPp),YD,
25     ((YB1,YS1)..(YBs,YSs)) //

      // create new node V to hold remainder of
      // prefix and subtrees //
30     V=MAKENODE((Pi+1..Pp),(D),((B1,S1)..
      (Bs,Ss)))
      // create new node W to hold common prefix
      // and new subtree //
35     W=MAKENODE((P1..Pi-1),(d),((Pi,V)))
      // replace pointer to X in Y with pointer to
      // W, then destroy X //
40     if Y=null
      then T=W
      else {YSy=W;output Y to disk}
45     KILLNODE(X);return(TRUE)
      end SUBSET

50 end PINSERT

```

55 D. Deletion of Nodes

The first step in deleting a node containing a given key which is to be deleted is to locate the node, which requires matching the key to the prefix completely, and determining whether there is data associated

with the node. Thereafter, the deletion of the node depends upon the number of branch characters, that is, the number of branches, dependent from the node.

In a first instance, there are no branch characters B_j in the node. That is, the node is a "leaf" node and there are no other keys in the search tree formed by this node and its subtrees. In this case, the node
 5 having the prefix which completely matches the key to be deleted is deleted and the subtree pointer and associated branch character which point to this node are removed from the parent node, that is, from the node containing the pointer to the node being deleted.

In the next case there is exactly one branch character in the node to be deleted. That is, the prefix matching the key occurs as the leading characters of at least one other key held in the search tree formed
 10 by the node to be deleted and its subtrees. The node to be deleted effectively operates as a placeholder for the key and all other branch points for other keys held in the tree formed of that node and its subtrees appear in the nodes of the subtrees dependent from that node.

This key is deleted by first deleting the data record associated with the node containing the matching prefix, that is, the data record pointed to by the D Field 36 pointer of that node. In the next step, however,
 15 the connection or branch connecting the single child node of the node to be deleted with the remainder of the tree must be preserved. This is accomplished by coalescing the prefix and branch character of the node to be deleted with the prefix of the child node, thereby creating a new node to replace both the node being deleted and the single child node dependent from that node. This new node, in effect, replaces the node that was deleted, and is pointed to by the branch pointer of the deleted nodes parent node that
 20 originally pointed to the deleted node.

This deletion of a node having a single branch is illustrated in Fig. 5, wherein the left hand drawing represents the original tree, and the right hand drawing the tree after the deletion of a node. As illustrated, the tree includes a root node T 49A with two branches and thus two branch characters, "B" and "H", with their associated pointers. The "B" branch pointer S_B goes to a branch which is not involved in the deletion
 25 operation, and which will not be discussed further. The branch dependent from the "H" branch character and pointed to by associated pointer S_H contains the keys "Hash", "HashTable", "HashTableFile" and "HashTableList". Node T49B contains the key "Hash", through branch character "H" in node T 49A and prefix "ash" in its PF 34, and has a single branch, dependent from branch character "T" through associated branch pointer S_T , and a data record reference through a D Field 36 pointer. Node 49B and key "Hash" are
 30 to be deleted from the tree in this example.

Node 49B's branch pointer S_T is to a node T 49C, which contains the prefix "able" and two branch characters, "L" and "F", with associated branch pointers S_F and S_L to nodes T 49D and T 49E respectively. Nodes T 49D and T 49E respectively contain prefixes "ist" and "ile" and D Field 36 pointers to data records.

In the deletion of node T 49B, the data record pointed to by T 49B's D Field 36 is located and deleted
 35 in the first step. Thereafter, T 49B and T 49C must be coalesced so as to preserve the keys and data record references of nodes T 49C, T 49D and T 49E, which are children of T 49B, and to maintain the links between the parent of T 49B, that is, T 49A, and T 49C, T 49D and T 49E. As illustrated in the right hand portion of Fig. 5, a new node T 49F containing the prefix "ashTable" is created, wherein this prefix is the
 40 coalition of prefixes "ash" from node T 49B and "Table" from node T 49C. Node T 49F has two branch characters, "L" and "F" from node T 49C, and associated branch pointers S_L and S_F to, respectively, nodes T 49D and T 49E. The branch pointer S_H of T 49A pointing to the original, deleted node T 49B now points to new node T 49F, so that the links from node T 49A through to nodes T 49D and T 49E are preserved.

In a final case of deletion of a node, the node to be deleted will have more than one branch character to
 45 child nodes, that is, the prefix of that node to be deleted will occur as the leading characters of at least two other keys held in the search tree formed from that node and its subtrees. In this instance, only the data is deleted from the node, by deleting the nodes D Field 36 pointer to the data record associated with the key to be deleted. It is necessary to retain the prefix and branch characters of the node as this node forms the branch point between the two or more keys held in the subtrees of the node.

50 Further description of the above node deletion operations will be found in the following exemplary Delete Program Listing C:

PROGRAM LISTING C - NODE DELETION

```

5  procedure PDELETE(T, (K1..Kk))
      // remove the key value (K1..Kk) from the
      // prefix search tree T.
      A tuple (i,d) is returned; i is false if K does
10     not exist.
      Otherwise i is true and d is the data record
      pointer //
15     if T=null then return(FALSE,null)
      X=T;Y=null;y=0;Z=null;z=0;j=0;n=0
      loop
20         input node X from disk
         let X be defined by p,s,(P1..Pp),D,
         ((B1,S1)..(Bs,Ss))
         // match the prefix to the leading characters
25         in the key //
         if k-n<p then return(FALSE,null)
         for i=1 to p do
30             n=n+1
             if Kn<>Pi then return(FALSE,null)
         end
35         // does the key match the prefix? //

```

40

45

50

55

```

    if n=k then {
        if D=null then return(FALSE,null)
        d=D
5       case
            :s=0:call LEAF(X,y,Y,z,Z)
            :s=1:call JOIN(X,y,Y)
10          :else:D=null
        end
        return(TRUE,d) )
15    // determine which node to process next.
        Search branch characters //
        n=n+1
20      z=y;y=j;j=1
        loop
            case
25          :j>s:return(FALSE,null)
            :Kn<Bj:return(FALSE,null)
            :Kn=Bj:exit
            :else:j=j+1
30          end
        forever
            Z=Y;Y=X;X=Sj
35    forever

```

40

45

50

55

LEAF

```

5  procedure LEAF(X,Y,Y,Z,Z)
      // The key has ended in a leaf node. We will de-
      //  lete this node, X, and the branch character,
      //   subtree pointer tuple, (BY,SY), in the par-
10  //   ent node, Y, which led us here. //
      // assume X, Y, and Z are already in memory
      let Y define p,s,(P1..Pp),D,((B1,S1)..
15      (Bs,Ss))
      let Z define Zp,Zs,(ZP1..ZPZp),ZD,((ZB1,
      ZS1)..(ZBZs,ZSZs)) //
20  // destroy node X //
      KILLNODE(X);
      // create new node W to hold contents of Y, minus
      //   one subtree //
25  if Y=null then (T=null;return)
      W=MAKENODE((P1..Pp),(D),((B1,S1)..(BY-1,
      SY-1),(BY+1,SY+1)..(Bs,Ss)))
30  // destroy node Y //
      KILLNODE(Y)
      // replace pointer to Y in Z with pointer to W //
35  if Z=null then (T=W;return)
      ZSZ=W;output Z to disk
      return
40  end LEAF

```

45

50

55

JOIN

```

5      procedure JOIN(X,y,Y)
      // The key has ended in a node with one subtree.
      We will create a new node to replace both this
      node, X, and the root node of the subtree
10     (B1,S1);
      // assume X, Y, and Z are already in memory
      let V define Vp,Vs,(VP1..VPVp),VD,
      ((VB1,VS1)..(VBVs,VSVs))
15     let X define p,s,(P1..Pp),D,((B1,S1)
      ..(BS,SS)) //
      let Y define Yp,Ys,(YP1..YPYp),YD,
20     ((YB1,YS1)..(YBYs,YSYs)) //
      // read next node, from subtree, into memory //
      V=S1;input node V from disk
25     // create new node W to hold contents of X
      plus V, minus one subtree //
      W=MAKENODE((P1..Pp,B1,VP1..VPVp),(VD),
      ((VB1,VS1)..(VBVs,VSVs)))
30     // destroy node V,X //
      KILLNODE(X);KILLNODE(V)
      // replace pointer to X in Y with pointer to W //
35     if Y=null then (T=W;return)
      YSy=W;output Y to disk
      return
40     end JOIN
      end PDELETE

```

45 While the invention has been particularly shown and described with reference to a preferred embodiment of the method and apparatus thereof, it will be understood by those of ordinary skill in the art that various changes in form, details and implementation may be made therein without departing from the spirit and scope of the invention as defined by the appended claims.

50 **Claims**

1. A prefix index tree structure for locating data records stored in a database in a data processing system through keys related to the information stored in the data records, each node of the tree comprising:
 55 a prefix field for storing a prefix string of length p comprised of the longest string of key characters shared by all subtrees of the node;
 a data record field for storing a reference to a data record whose key is completed by the prefix string; and,
 when the prefix string is a prefix of keys stored in at least one subtree of the node, a branch field for each

- distinct $p + 1^{\text{st}}$ key character in the keys of the subtrees, wherein each distinct $p + 1^{\text{st}}$ key character is a branch character and each branch field comprises
- a branch character field for storing the $p + 1^{\text{st}}$ character of a key, and
 - a branch pointer field for storing a reference to a node of a subtree containing at least one key whose $p + 1^{\text{st}}$ character is the branch character.
- 5 2. The node of the prefix index tree structure of claim 1, wherein each node further comprises:
- a field for storing a number equal to the number of key characters in the prefix string, and
 - a field for storing a number equal to the number of branch fields in the node.
3. A method for constructing a prefix index tree structure for locating data records stored in a database in a data processing system through keys related to the information stored in the data records, comprising, for
- 10 each node of the tree, the steps of:
- determining a prefix string of length p that is the longest string of key characters shared by all subtrees of the node,
 - storing the prefix string in a prefix field of the node;
- 15 when there is a data record whose key is completed by the prefix string,
- storing a reference to a data record whose key is completed by the prefix string in a data record field of the node; and,
 - when the prefix string is a prefix of keys stored in at least one subtree of the node,
- 20 determining the branch characters for all of the keys stored in each subtree, wherein each branch character is a distinct $p + 1^{\text{st}}$ character of a key contained in a subtree of the node, and
- creating a branch field for each branch character, and
 - storing the corresponding branch character in a branch character field of the branch field, and
 - storing a reference to a node of a subtree containing at least one key whose $p + 1^{\text{st}}$ character is the branch character in a branch pointer field of the branch field.
- 25 4. In a prefix index tree of claim 1,
- a method for searching the prefix index tree to locate a data record using search keys related to the information stored in the data records, comprising the steps of:
- comparing a search key of length k greater than p to the prefix string of a node,
 - when there is no match between the search key and the prefix string,
- 30 terminating the search,
- when there is a complete match between the search key and the prefix string,
 - reading the reference from data record field of the node to determine the location of the data record whose key corresponds to the search key, and,
 - when the initial p characters of the search key match the prefix string,
- 35 compare the $p + 1^{\text{st}}$ character of the search key to the branch characters of the branch fields of the node, and
- when there is no match between the $p + 1^{\text{st}}$ of the search key and a branch character,
 - terminating the search, and
 - when there is a match between the $p + 1^{\text{st}}$ of the search key and a branch character,
- 40 reading from the branch pointer field of the branch field the reference to the subtree node containing a key whose $p + 1^{\text{st}}$ character matches the $p + 1^{\text{st}}$ character of the search key, and
- repeating the above steps with respect to the node referenced by the branch pointer field.
5. In a prefix index tree of claim 1,
- a method for inserting a new key into a node of the prefix index tree when there is a mismatch between the key and the prefix string that occurs before the end of both the key and prefix string, comprising the steps of:
- creating a first new node containing,
 - in its prefix field the portion of the key occurring after the key character that caused the match of key and original prefix string to fail, and
- 50 in its data record field a reference to the data record associated with the new key,
- creating a second new node containing,
 - in its prefix field the portion of the original prefix string occurring after the original prefix character that caused the match of key and original prefix string to fail, and
 - in its data record and branch fields the contents of the data record and branch fields of the original node,
- 55 and
- creating a third new node containing,
 - in its prefix field the portion of the original prefix string which matched with the key,
 - a first branch field having

- in its branch character field the key character which caused the match between the original prefix string and key to fail, and
in its branch pointer field a reference to the first new node, and
a second branch field having
- 5 in its branch character field the character of the original prefix string which caused the match between the original prefix string and key to fail, and
in its branch pointer field a reference to the second new node.
6. In a prefix index tree of claim 1,
a method for inserting a new key of length k into a node of the prefix index tree when the prefix string of
10 the node is of length p less than k and matches the initial p characters of the new key and the node contains no branch character matching the p + 1st character of the key, comprising the steps of:
creating a new node containing
in its prefix field the portion of the new key following the p + 1st character of the new key, and
in its data record field a reference to the data record associated with the new key, and
15 in the original node,
adding a new branch field containing
in its branch character field the p + 1st character of the new key, and
in its branch pointer field a reference to the new node.
7. In a prefix index tree of claim 1,
20 a method for inserting a new key of length k into a node of the prefix index tree when the prefix string of the node is of length p greater than k and the initial p characters of the new key match the prefix string, comprising the steps of:
creating a first new node containing
in its prefix field the portion of the original prefix string following the k + 1st of the original prefix string, and
25 in its data record and branch fields the contents of the data record and branch fields of the original node, and
creating a second new node in replacement for the original node, containing
in its prefix field the portion of the original prefix string that was matched by the search key, and a branch field, containing
30 in its branch character field the k + 1st of the original prefix string, and
in its branch pointer field a reference to the first new node.
8. In a prefix index tree of claim 1,
a method for deleting a key from the tree, comprising the steps of:
determining the node containing the key to be deleted and the number of branch characters of the node,
35 when there are no branch characters in the node,
deleting the node, and
deleting the branch character and branch pointer to the deleted node from the branch field of the parent node of the deleted node,
when the node contains more than one branch character,
40 deleting the data record pointer referencing the data record associated with the key to be deleted, and
when the node contains one branch character,
locate the child node referenced by the branch pointer of the single branch field of the node,
create a new prefix string for the node by coalescing the original prefix string and the prefix string of the child node,
45 delete the original single branch field from the node, and
write the branch fields and data record field from the child node into the branch fields and data record field of the node.

50

55

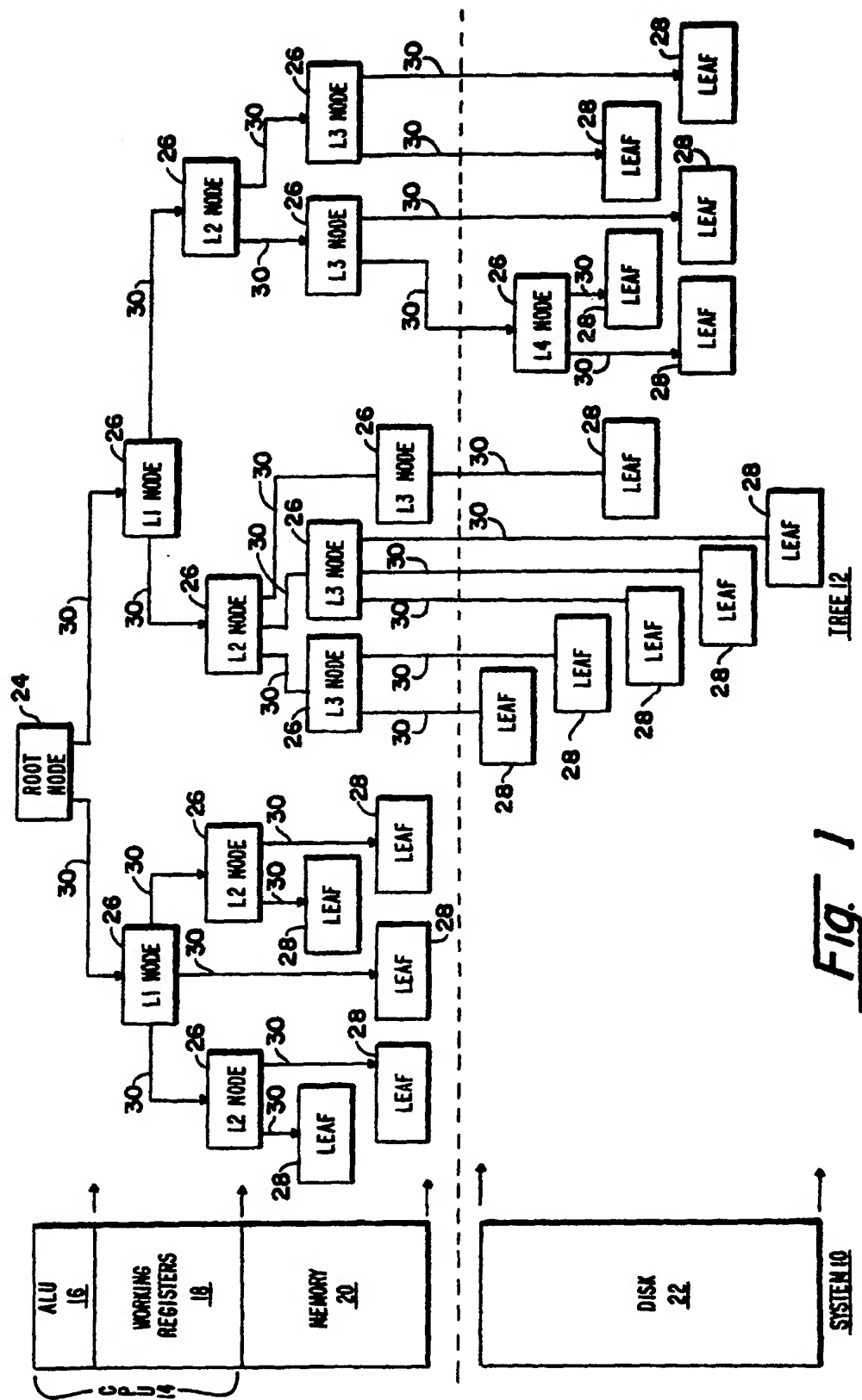


Fig. 1

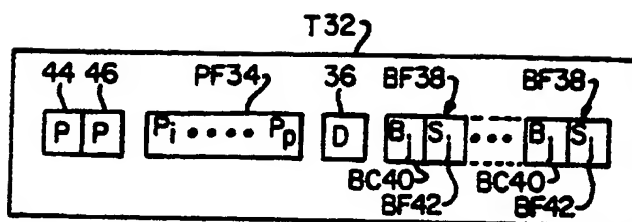


Fig. 2

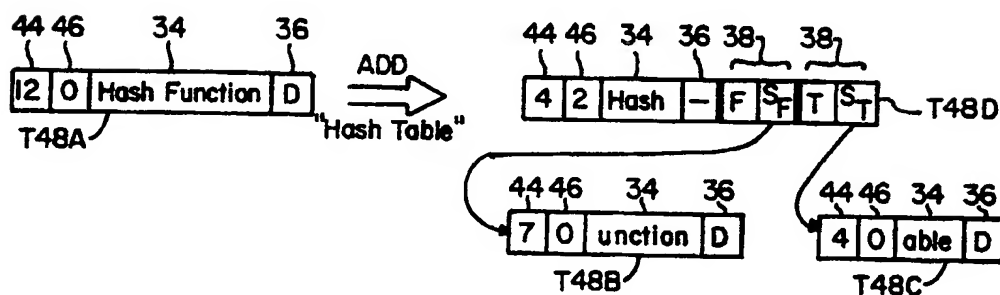


Fig. 4A

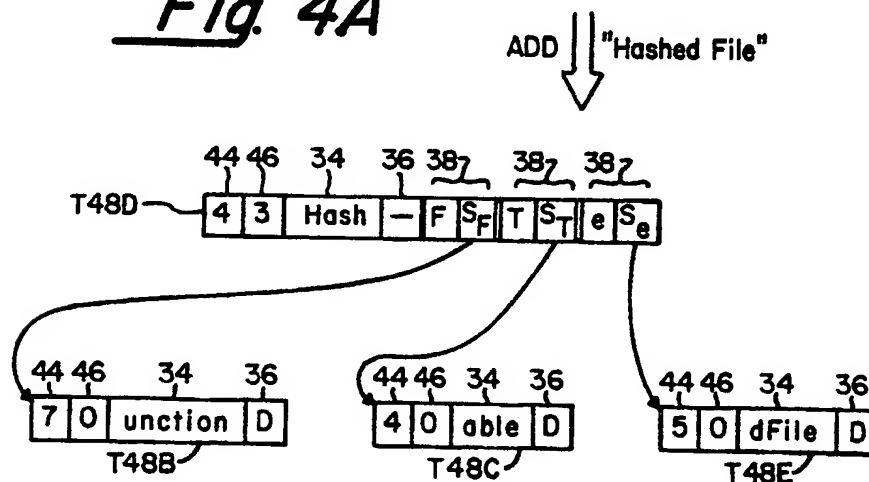


Fig. 4B

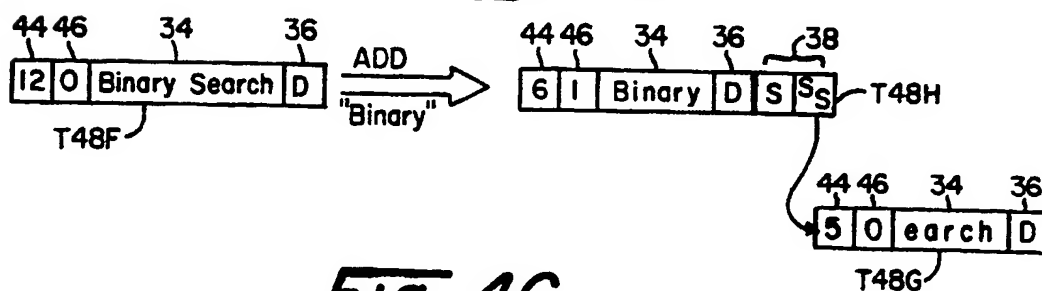


Fig. 4C

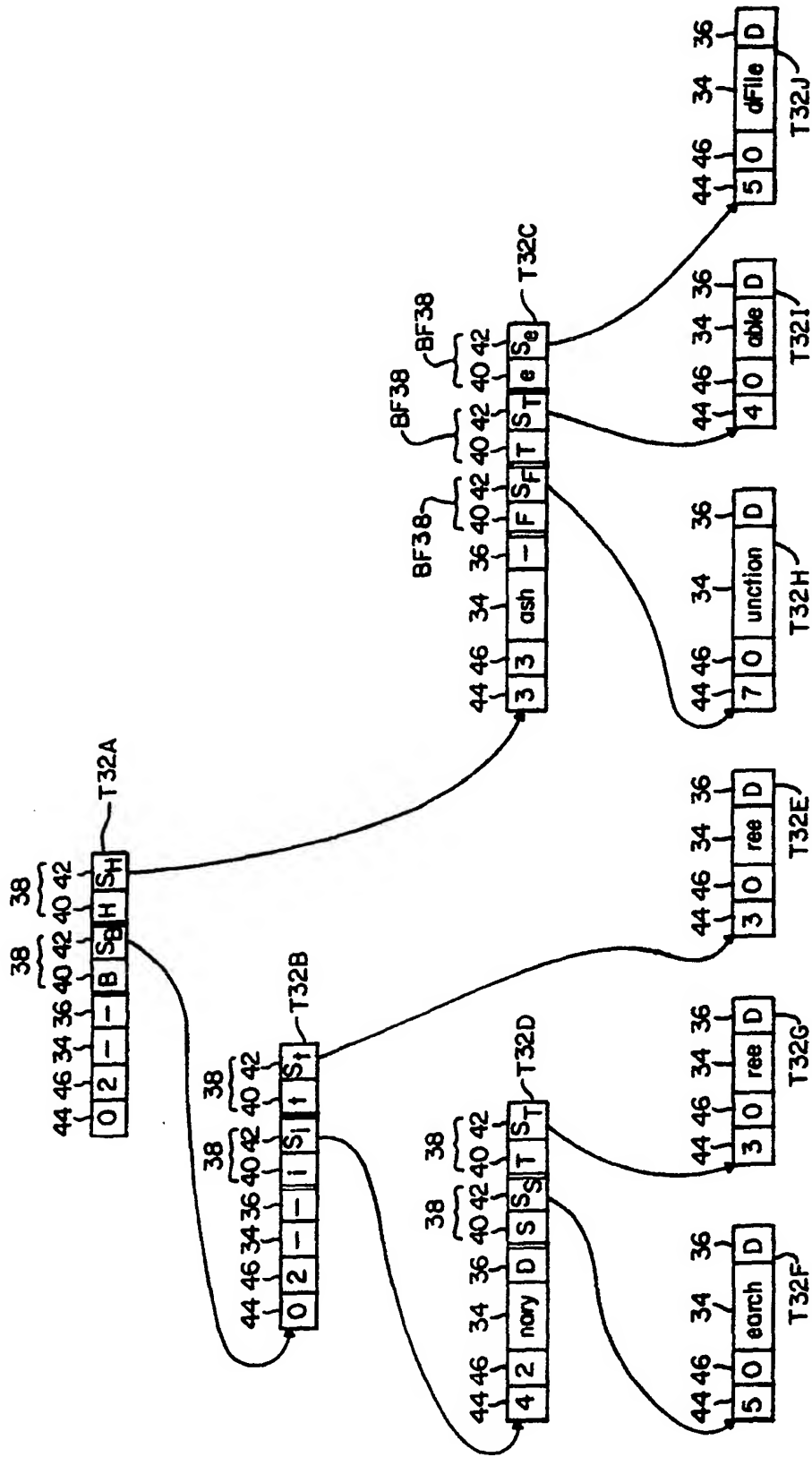


Fig. 3

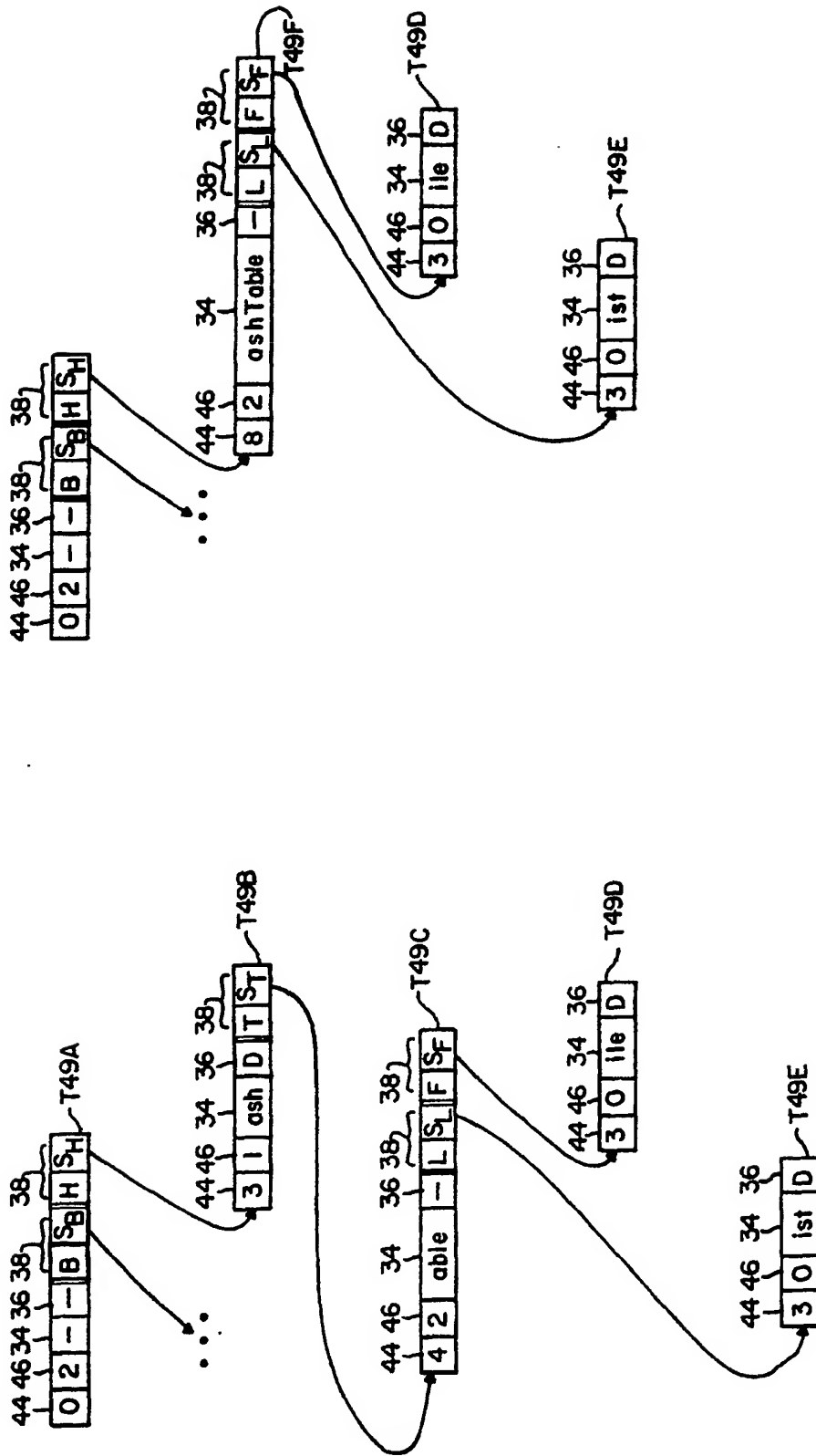


Fig. 5